Untangling the Knot: Automated Component Refactoring Assistant

Presenters: James Ivers
Ipek Ozkaya

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM19-0692

Agenda

I. Introductions (15-30 min)

- SEI introductions/background
- Novetta introductions/background

II. Overview of our research project (1-1.5 hr)

III. Overview of how SEI would like to use Novetta code (1 hr)

IV. Next steps (30 min)

Agenda

I. Introductions (15-30 min)

II. Overview of our research project (1-1.5 hr)

- Overall objectives and approach
- Where we are today
- Demo and present results to date

III. Overview of how SEI would like to use Novetta code (1 hr)

IV. Next steps (30 min)

Untangling the Knot: Automated Component Refactoring Assistant

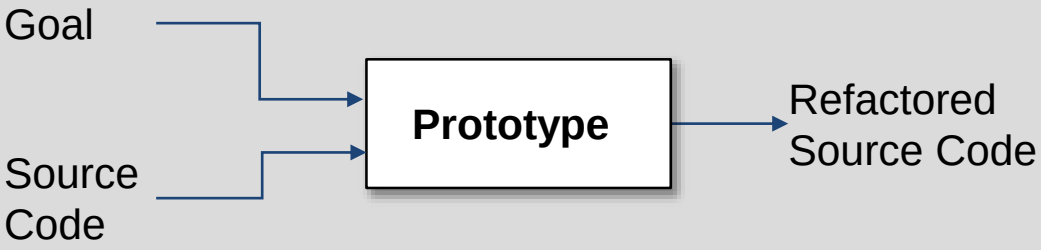
Problem: Harvesting or replacing large-scale software components in poorly structured software is a slow, labor-intensive activity and (often cost prohibitive).

Solution: Create an automated component refactoring assistant that recommends architectural refactorings and implements them through code transformations.

Approach:

- Formalize the goal and use it to drive recommendations.
- Derive and expand refactorings for architecture scale changes.
- Adapt search-based algorithms to generate suitable recommendations.

Project Objective



Targets: Our prototype will make code changes that

- Resolve at least 75% of all problematic couplings and
- Require no more than 10% developer time compared to a manual effort.

Advance the state of the practice by increasing automation and decreasing decision burden on developers.

Advance the state of the art by focusing on solving specific problems rather than providing general improvements.

Impact

An example: A contractor recently estimated the development work alone to rearchitect a navigation capability to modularize components and reduce coupling with the hardware platform.

- 14K hours; roughly $1.15M[1]

Our work would reduce the development effort to

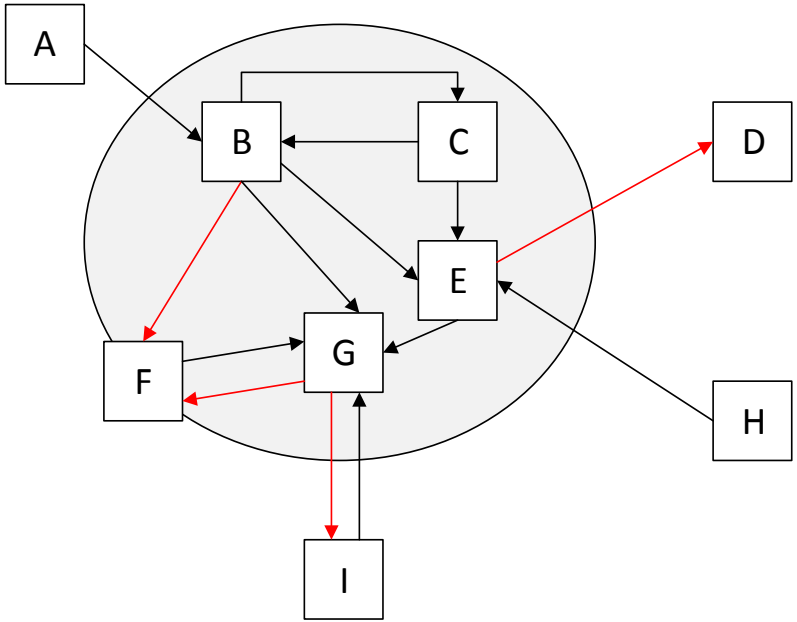
- 4.5K hours; roughly $370K
- **Savings:** 9.5K developer hours and roughly $780K

Recall our **objective**

- Resolve at least 75% of all problematic couplings and
- Require no more than 10% developer time compared to a manual effort.

[1] All costs are based on an average burdened rate of $150K per year [Clark 2017]

Core Concepts



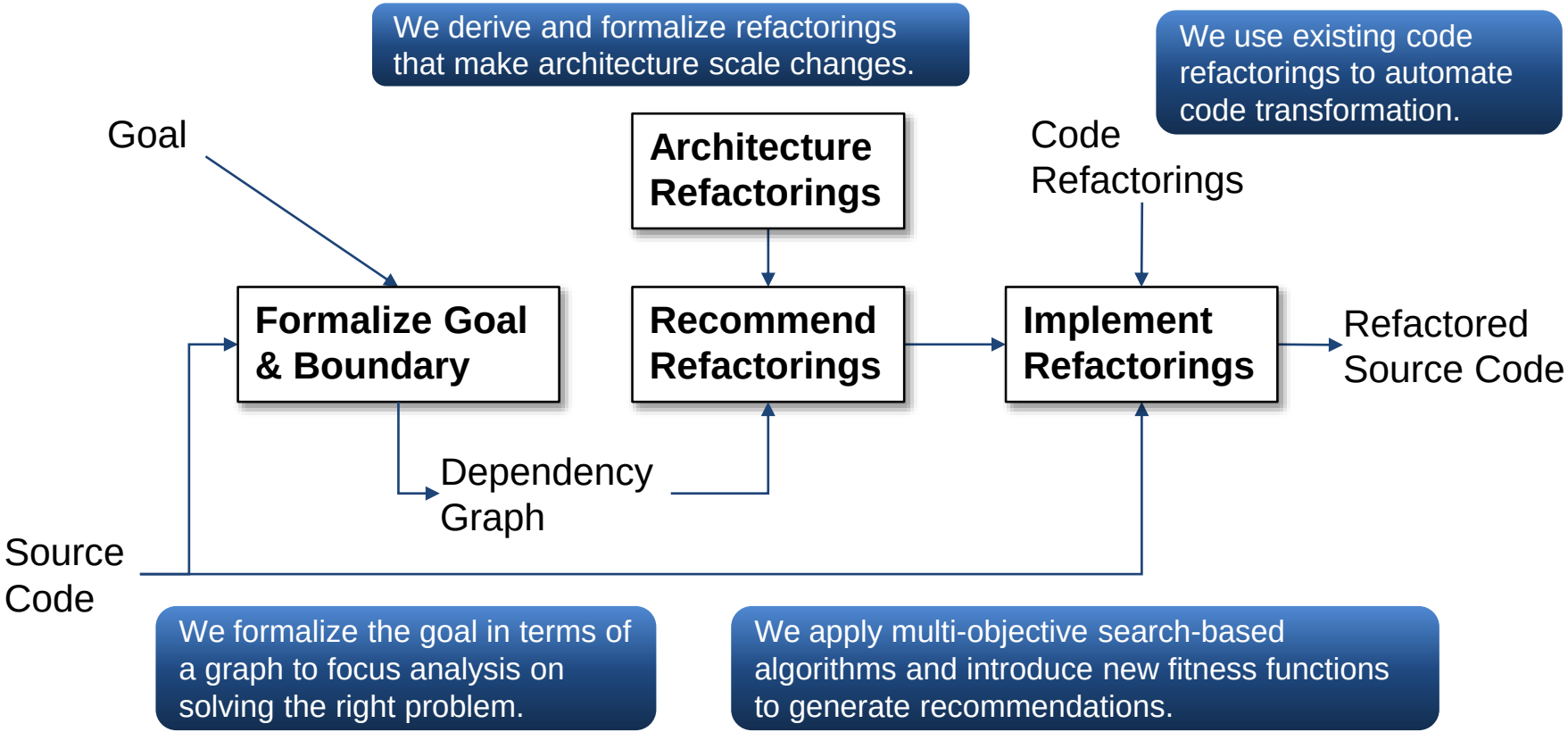
Harvesting and replacing components both rely on managing the **dependencies** among software **elements**.

Programming languages include dozens of types of elements and dependencies.

- Elements include classes, interfaces, variables, and methods
- Dependencies include calls, reads, inherits from, and implements

The challenge is to identify and resolve **problematic couplings** (i.e., the subset of dependencies that inhibit a goal).

Technical Underpinnings



Formalize Goal and Boundary

The user is trying to accomplish a **goal**

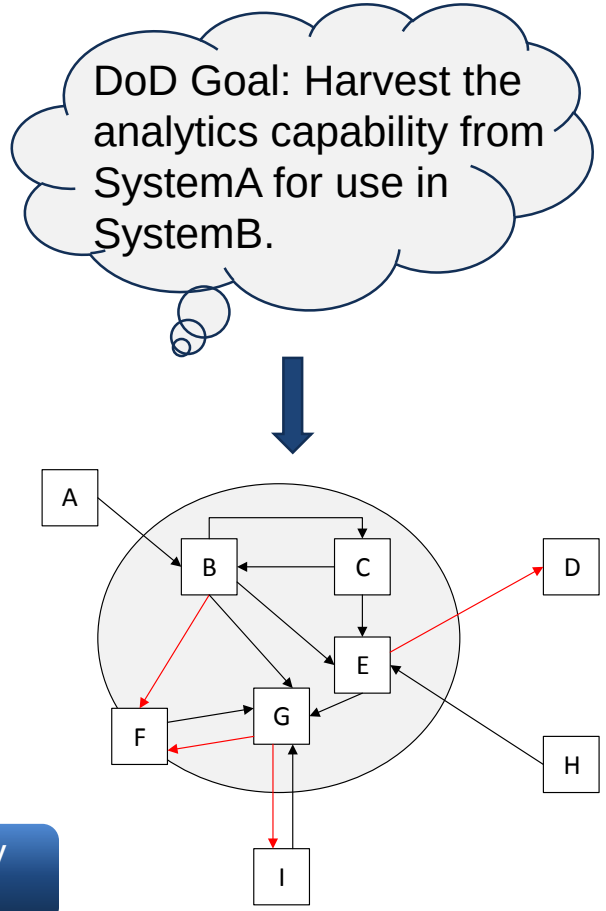
- Scenario: harvest, replace, …
- Scope: what code implements the capability

Elicit this information interactively with a developer

- Augmented by structural code analysis and graph query to refine **boundary**

Create a **heterogeneous graph** over which

- Semantic intent is formalized as allowable dependencies across the boundary
- Problematic couplings are machine enumerable



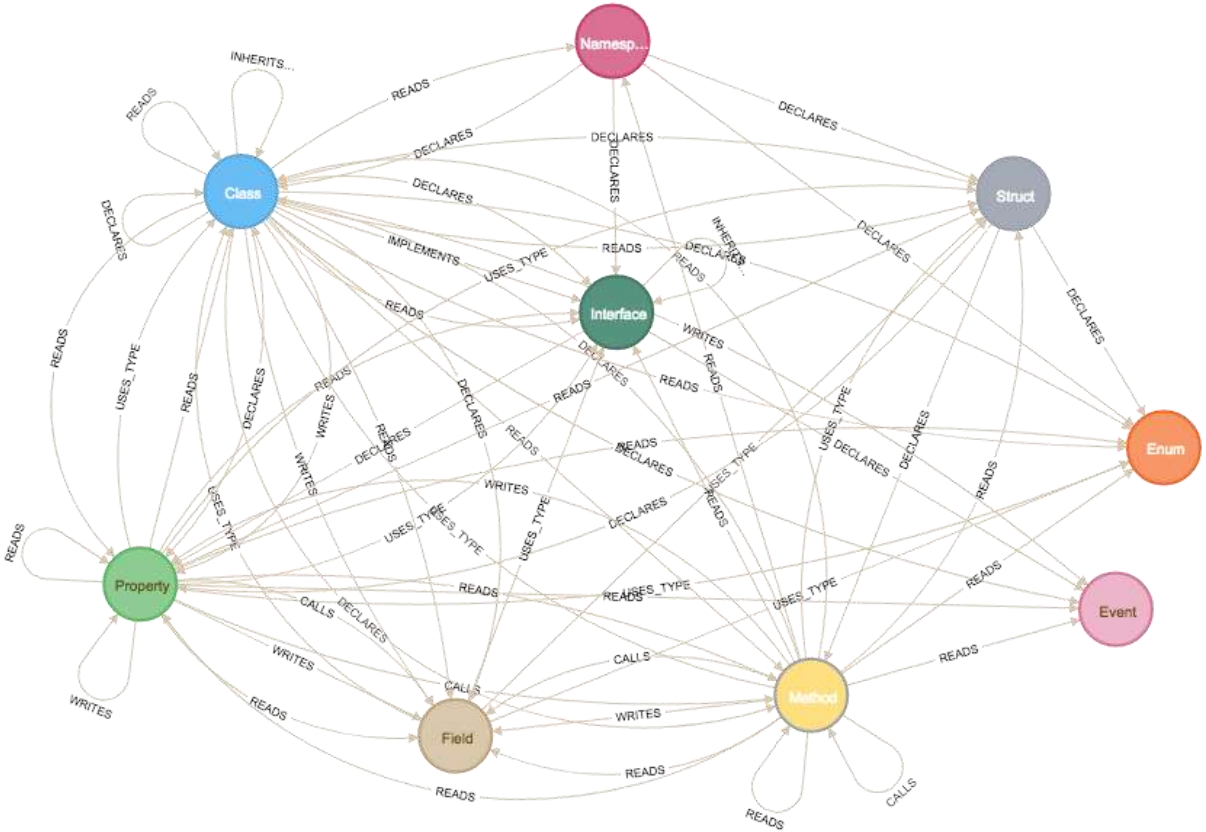
This formalization shifts recommendations from general code quality improvement to solving specific problems.

Goal and Boundary - 1

Created a Neo4j graph representation of C# code

Sample graph sizes

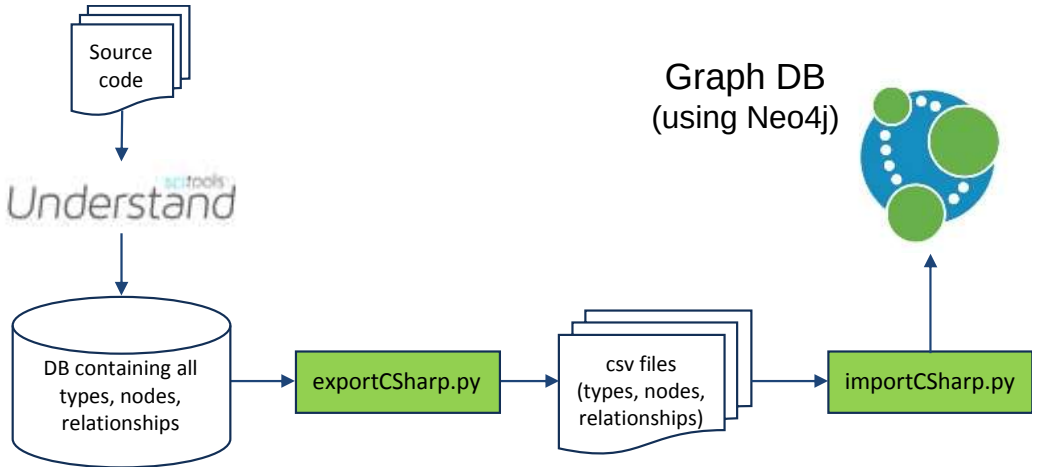
- Duplicati:
  - 226K code lines
  - 10,194 nodes and 49,620 relations
- MissionPlanner:
  - 939K code lines
  - 81,790 nodes and 587,542 relations



Goal and Boundary - 2

We created a tool chain to build these graphs

- SciTools Understand is a commercial code analysis tool that provides an open API
- Neo4j is a widely used graph platform with open source and enterprise editions
- Our scripts
  - Extract the relevant data from Understand
  - Reorganize the data to support our uses
  - Use the data to populate a Neo4j database



Our prototype

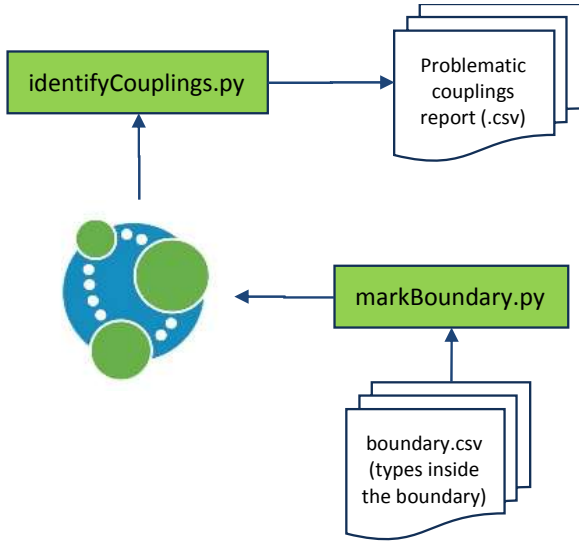
Goal and Boundary - 3

We create a simple means to allow users to describe their goal and boundary

- Defined six types of goal; a user simply picks one
- Defined a format for a user to identify what code lies within boundary
- Formalized goal-specific meanings of problematic couplings
  - Essentially, problematic couplings are dependencies that cross a boundary in the "wrong" direction
  - Easily expressed as Cypher queries

We extended our tool chain with scripts to

- Mark a boundary on our graph by "coloring" nodes
- Enumerate all problematic couplings in a graph given a goal and boundary

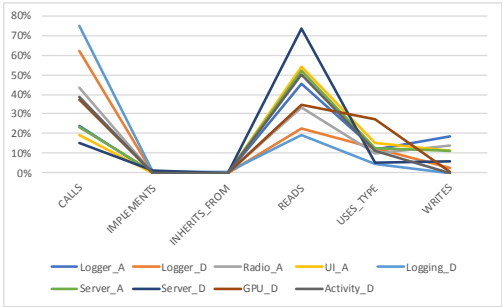


Goal and Boundary - 4

At this stage, we can already use our tools to assess the complexity involved in harvesting or replacing a collection of code.

We have defined nine scenarios across four open source case studies and are analyzing them to understand the distribution of problematic couplings by relation type, destination type, etc.

		Problematic Couplings						
Project	Scenario	CALLS	IMPLEMENTS	INHERITS_FROM	READS	USES_TYPE	WRITES	Total
MissionPlanner	Logger_A	515	0	1	982	255	403	2156
MissionPlanner	Logger_D	25	0	0	9	5	1	40
MissionPlanner	Radio_A	135	0	0	103	30	43	310
MissionPlanner	UI_A	2557	2	2	7269	2085	1493	13408
Duplicati	Logging_D	448	4	2	114	28	0	596
Duplicati	Server_A	105	3	0	235	56	52	451
Duplicati	Server_D	65	4	0	320	22	24	435
ConvNetSharp	GPU_D	529	0	1	495	384	7	1416
SharpCaster	Activity_D	10	0	0	13	3	0	26



Architecture Refactorings

An architecture refactoring is a code transformation that realizes an architecture improvement.

Derive architecture refactorings from existing sources (top down) and analysis of problematic couplings in case studies (bottom up).

Formalize for each architecture refactoring

- Applicability rules, as predicates over the graph
- Effects, as graph transformations

Candidate refactorings from Fowler:

- *Change Bidirectional Association to Unidirectional*
- *Extract Class*
- *Extract Interface*
- *Hide Delegate*
- *Move Method*
- *Move Field*
- *Pull Up Method*
- …….

This formalization is necessary for automation. The set of precisely defined refactorings will advance the state of the art.

Refactorings - 1

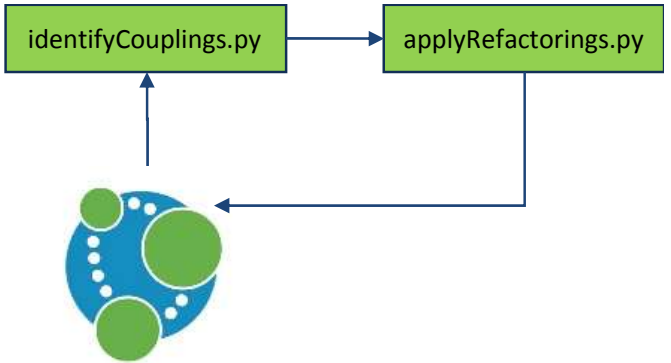
Defined four refactorings as an initial set

- Move Class
- Move Interface
- Move Static Field
- Move Static Method

Formalized each in terms of preconditions over and transformations of the graph

We extended our tool chain with scripts to implement each

We expect to create many more over the next year



MVP Version of Move Class

Preconditions

Check the properties of the nodes and the relationship of the PC

- (pc.dest.type == 'Method') *v* (pc.dest.type == 'Field') *v* (pc.dest.type == 'Property') *v* (pc.dest.type == 'Class')
  - members of the destination nodes of the pc are restricted to methods, fields, classes, or properties
- (pc.type == 'CALLS') v (pc.type == 'READS') v (pc.type == 'WRITES') *v* (pc.type == 'USES_TYPE')
  - list of relationships that this MVP refactoring is restricted to

Is the dest a class or a member of a class, not looking at structs for this MVP, structs can have methods, fields as members too.

- (existsRelation ('DECLARES', 'Class', pc.dest) == true) v (pc.dest.type == 'Class')

The class to be moved cannot be abstract

- if (pc.dest.type == 'Class')
  - classToBeMoved = pc.dest
- else
  - classToBeMoved = getRelation ('DECLARES', 'Class', pc.dest).src
- propertyof (classToBeMoved, 'abstract') == false

The class to be moved cannot be nested within another class

- existsRelation ('DECLARES', 'Class', classToBeMoved) == false

Transformation

Key idea: transformation will have to change labels on the class and all members declared by the class (transitive through nested types)

Reverse the label

- if (pc.dest.type == 'Class')
  - classToBeMoved = pc.dest
- else
  - classToBeMoved = getRelation ('DECLARES', 'Class', pc.dest).src
- reverse_scenario_label (classToBeMoved, scenario_label)

Reverse the label for all the members of classToBeMoved

- memberDecls = getRelationsFrom ('DECLARES', classToBeMoved)
- for each decl in memberDecls
  - member = decl.dest
  - reverse_scenario_label (member, scenario_label)
  - // recurse here: now treat member as class, check for all things it declares, relabel them, and recurse until no further child declarations

Refactorings - 2

Data on application of each refactoring gives us insight into

- How useful are our MVPs?
- Which restrictions in our MVPs should we loosen next?

				Exclusions										
Project	Scenario	Total PCs	# Applicable	Not a Method	Not a CALL	Method not Static	Dest Scope Not a Class	Orig Scope Not a Class	Too Many Relations to Method	Method Not in a Single File	Too Many Relations from Method	Dest Scope Inherits	Orig Scope Inherits	Orig Scope Implements Interface
MissionPlanner	Logger_A	2156	1	1641	0	429	0	2	81	0	2	0	0	0
MissionPlanner	Logger_D	40	0	15	0	11	0	0	12	0	2	0	0	0
MissionPlanner	Radio_A	310	0	176	0	90	0	0	44	0	0	0	0	0
MissionPlanner	UI_A	13410	11	10853	1	1961	0	49	472	0	61	2	0	0
Duplicati	Logging_D	596	0	145	3	62	0	0	382	0	4	0	0	0
Duplicati	Server_A	451	1	346	0	56	0	0	39	0	9	0	0	0
Duplicati	Server_D	435	0	371	0	48	0	0	16	0	0	0	0	0
ConvNetSharp	GPU_D	1416	0	887	0	446	0	0	81	0	2	0	0	0
SharpCaster	Activity_D	26	0	16	0	8	0	0	1	0	1	0	0	0

Our initial Move Static Method, for example, was much too restrictive.

Refactorings - 3

As we add new refactoring and loosen restrictions on existing refactorings, we create more options (potential solutions) for each problematic coupling. More options should increase the effectiveness of the genetic search as we start that phase of the project.

				# Appplicable Refactorings				
Project	Scenario	Total PCs	% w/ RF	0	1	2	3	4
MissionPlanner	Logger_A	2156	78%	473	1682	1	0	0
MissionPlanner	Logger_D	40	100%	0	40	0	0	0
MissionPlanner	Radio_A	310	100%	1	309	0	0	0
MissionPlanner	UI_A	13410	68%	4296	9100	14	0	0
Duplicati	Logging_D	596	93%	39	557	0	0	0
Duplicati	Server_A	451	94%	27	423	1	0	0
Duplicati	Server_D	435	93%	30	403	2	0	0
ConvNetSharp	GPU_D	1416	42%	816	600	0	0	0
SharpCaster	Activity_D	26	100%	0	26	0	0	0

We expect to see regular progress in this area over the next year, but we already have at least one viable option for ~70% of all problematic couplings in our case studies.

Recommend Refactorings

Apply multi-objective search to generate recommendations.

- Representation – the heterogeneous dependency graph
- Operations – the architecture refactorings
- Algorithms – starting with non-dominated sorting genetic algorithms (NSGA-II)
  - Multiple objectives expressed through fitness functions

We will

- Define a fitness function that incorporates the goal
- Evaluate different combinations of additional fitness functions, such as semantic coherence, size of changes required, and modifiability metrics
- Compare local, global, and sub-graph computation

This will expand the applicability of search-based approaches to a wider range of software engineering problems.

Search - 1

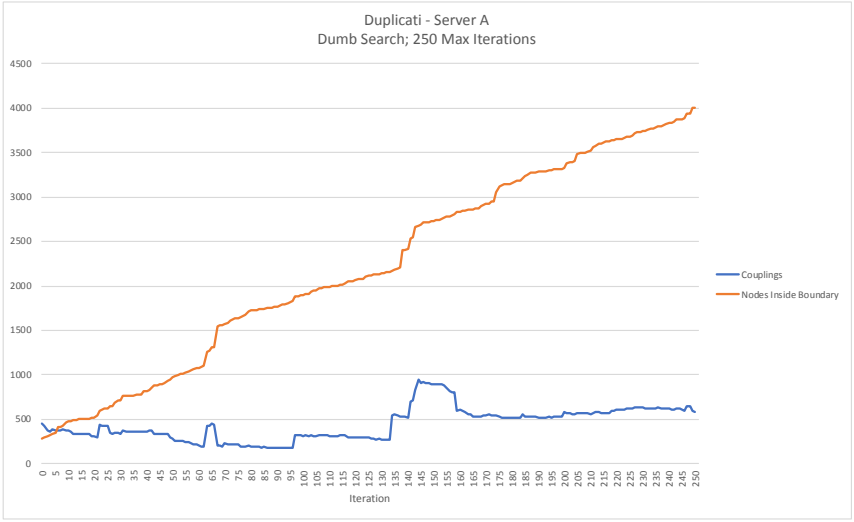
Engaging with campus collaborators (Claire Le Goues and Chris Timperley) to work on search algorithms

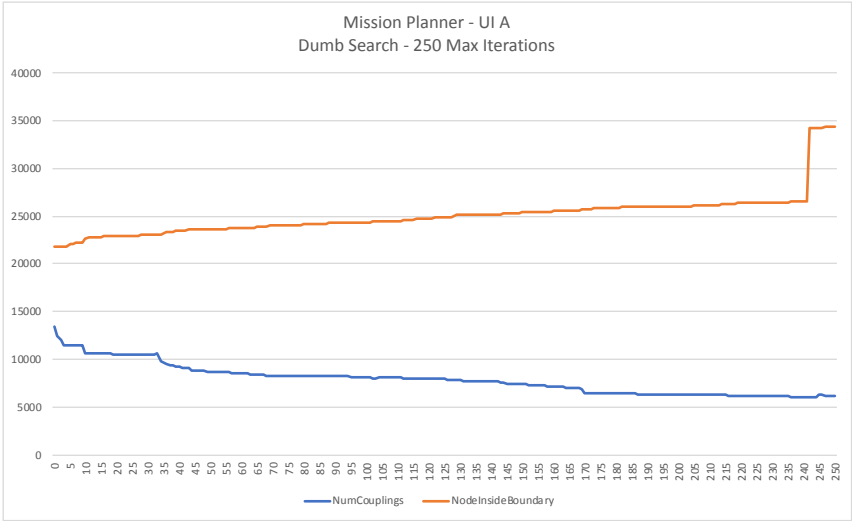
- Sketches of search strategy
- Initial options for fitness functions, such as
  - Minimizing problematic couplings
  - Minimizing addition of code to destination
  - Minimizing code changes
  - Minimizing creation of unrealized interfaces
  - Maximizing semantic coherence
- Candidate heuristics to apply in sampling and selection

Search - 2

Rudimentary search implemented to build out infrastructure

- Limited options available at present
- Measures against crude fitness measures





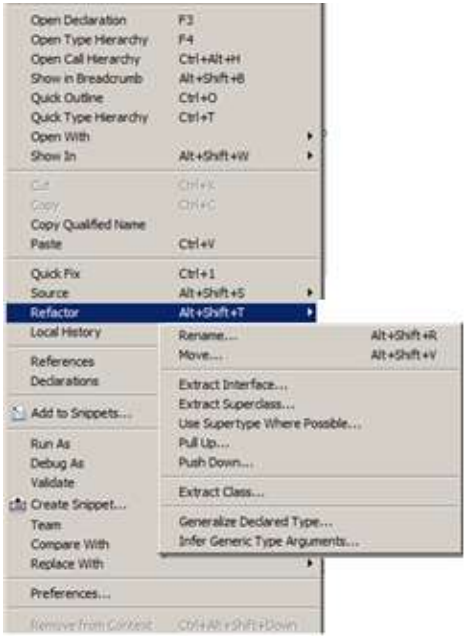
Implement Refactorings

Implement architecture refactorings as a sequence of code transformations existing in IDEs and their plugins

- Common code refactorings have mature implementations
- Test to confirm correctness (samples and regression over case studies)

Address developer confidence

- Provide a preview of changes
- Organize the preview to group changes
- Generate an audit trail



Eclipe IDE

This implementation closes the loop, enabling experimental application including validation of results and developer acceptance.

Agenda

I. Introductions (15-30 min)

II. Overview of our research project (1-1.5 hr)

III. Overview of how SEI would like to use Novetta code (1 hr)

- How it would fit into our experiments and the kind of data we're looking for
- Opportunities for Novetta staff to be involved and/or keep up with our progress
- Examples of ways that SEI might be able to refer to findings (e.g., how to anonymize identity)
- How SEI will handle data and pre-release review
- Any Novetta concerns

IV. Next steps (30 min)

Agenda

I. Introductions (15-30 min)

II. Overview of our research project (1-1.5 hr)

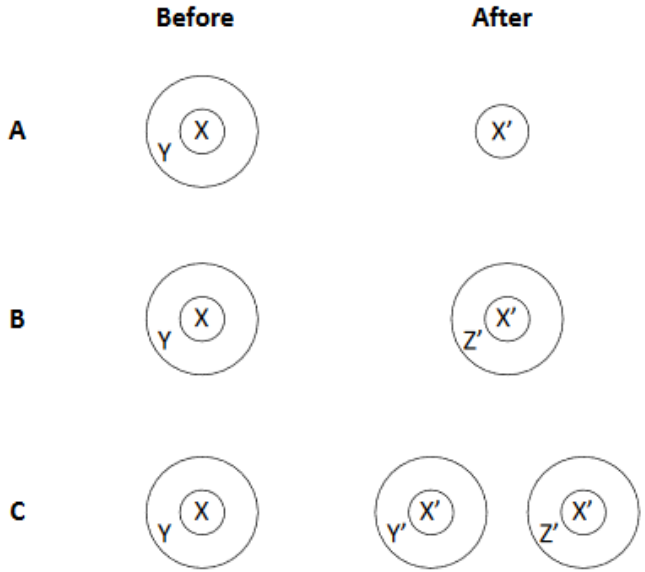
III. Overview of how SEI would like to use Novetta code (1 hr)

IV. Next steps (30 min)

- How to get a copy of the code (CD, AMRDEC SAFE, other?)
- Quick orientation to code structure
- Scenario suggestions
- Cadence for staying in touch

BACKUP

Boundary Scenarios - Harvest



(A) **Harvest (**X) for use only in an unknown context.
Example: harvesting for deployment as a standalone service.
Outcome: The original context (Y) does not need to work with the refactored (X).

(B) **Harvest** (X) for use in a known context (Z).
Example: harvesting for use in another system or as a core asset in a planned software product line or clone and own scenarios.
Outcome: Both (X)and (Z) will change during refactoring. The original context (Y) does not need to work with the refactored (X).

(C) **Harvest** (X) for use in a known context (Z) while retaining use of a common (X) in the original context (Y).
Example: promote use of a common service without forcing (Y) to evolve to fully adopt a service model.
Outcome: (X), (Y), and (Z) will all change during refactoring.

Harvest: The extraction cares about the code within the boundary, X
Replace: The extraction does not care about the code within the boundary, X

Boundary Scenarios – Replace and Cleanup

(D) **Replace** (X) with an unknown component.
Example: excise a piece of proprietary code, leaving a cleaner interface in the containing system (Y') for a yet to be determined replacement.
Outcome: Changes to (Y), (X) is thrown away.

(E) **Replace** (X) with an known component (W).
Example: replace a component with an alternate from another supplier.
Outcome: Both the new component (W) and the system context (Y) can change during refactoring.

(F) **Cleanup** the boundary between a component (X) and the rest of the system (Y). (Note that this scenario also represents state-of-the-art.)
Example: In preparation for a future Harvest or Replace, or to reduce the coupling, which in turn reduces maintenance complexity.
Outcome: Changes to both (X) and (Y), but (X) is not expected to be completely decoupled from the rest of the system.

Harvest: The extraction cares about the code within the boundary, X
Replace: The extraction does not care about the code within the boundary, X

Identifying Problematic Couplings

The formal definition of a problematic coupling is relative to the scenario. The following Cypher queries provide our initial definitions.

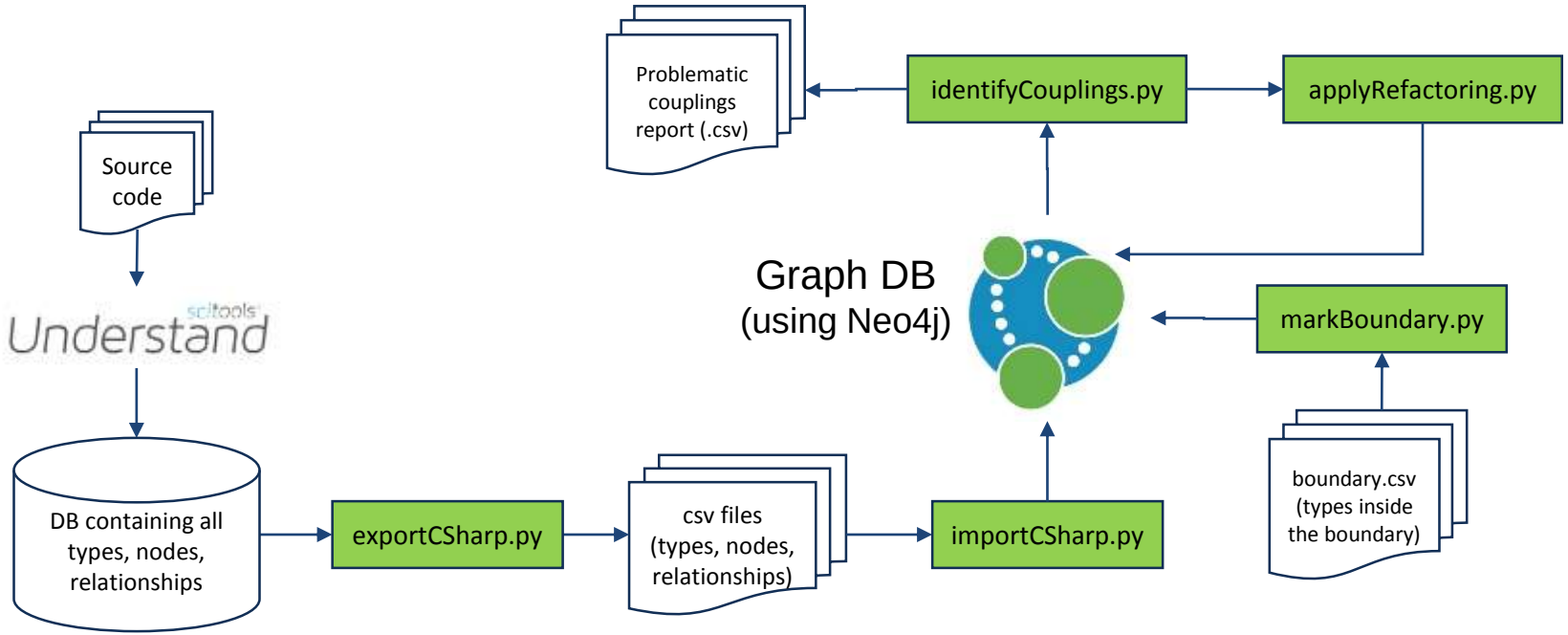
Scenario A (Harvest)

```
MATCH p=(o:InBoundary)-[r:CALLS|IMPLEMENTS|INHERITS_FROM|READS|USES_TYPE|WRITES]-
>(i:OutBoundary)
RETURN p
```

Scenario D (Replace)

```
MATCH p=(o:OutBoundary)-[r:CALLS|IMPLEMENTS|INHERITS_FROM|READS|USES_TYPE|WRITES]-
>(i:InBoundary)
RETURN p
```

Tool Chain (So Far)



Our prototype